



Compositional model checking of an SDN platform

Abdul Majith, Ocan Sankur, Hervé Marchand, Thai Dinh

► To cite this version:

Abdul Majith, Ocan Sankur, Hervé Marchand, Thai Dinh. Compositional model checking of an SDN platform. DRCN 2021 - 17th International Conference on the Design of Reliable Communication Networks, Apr 2021, Milan, Italy. pp.1-8. hal-03229532

HAL Id: hal-03229532

<https://inria.hal.science/hal-03229532>

Submitted on 19 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compositional model checking of an SDN platform

Abdul Majith

Univ Rennes, Inria, CNRS
Rennes, France

abdul-majith.noordheen@inria.fr

Ocan Sankur

Univ Rennes, Inria, CNRS
Rennes, France

ocan.sankur@irisa.fr

Hervé Marchand

Univ Rennes, Inria, CNRS
Rennes, France

herve.marchand@inria.fr

Dinh Thai Bui

Nokia Bell Labs France
Nozay, France

dinh_thai.bui@nokia-bell-labs.com

Abstract—Software-Defined Network (SDN) technology provides the possibility to turn the network infrastructure into a dynamic programmable fabric capable of meeting the application needs in real-time. Thanks to the independence of the control plane from the data plane, the control entity, generally called as controller, has also the flexibility to implement proprietary complex algorithms. Within such a dynamic and complex environment, this document advocates for applying formal verification methods and more precisely composition model checking to ensure the correct behavior of the overall SDN system at design phase. To illustrate this purpose, it proposes to build different comprehensive formal models of a typical SDN platform selected here as a study object. Thorough performance results related to each model are provided and discussed. Thanks to such formal verifications, it is possible to pinpoint issues such as the one regarding network isolation within a complex SDN architecture. Although dealing with formal methods, this document attempts to strike a balance between theory, experimental work and network architecture discussion.

Index Terms—SDN, slice, compositional checking, verification, isolation, safety

I. INTRODUCTION

The flexibility of Software-Defined network (SDN) together with the implementation of new complex network algorithms raises new challenges in term of validation and verification. Formal verification methods which are capable of exhaustively checking the behaviors of a given system have been applied in the setting of SDNs (see Section II). We also advocate for applying formal verification methods to ensure the correct behavior of an SDN platform especially during the design phase. More precisely, we demonstrate that *compositional model checking* is well-adapted to model and verify SDN architectures with all their different layers or planes, namely the data plane, the control plane and the management plane. For this purpose, we propose to build complete models of a typical SDN architecture taken here as a study object. Compositional modeling allows for designing and verifying each layer separately from others leading to a more modular and more scalable verification mechanism. Our contribution is fourfold. First, we present a comprehensive formal model of the SDN platform published in [1]. We model respectively devices and their mobility, switches and the network topology on the data plane, controllers on the control plane and managers on the management plane. We model communications between components of the same planes (i.e. between devices and switches) as well as those in different planes. Second, we demonstrate that compositional reasoning is well-adapted

to model-check SDN by decomposing the overall complex system verification into the sum of simpler verifications of the different components. We perform both a vertical (e.g. between planes) and a horizontal (e.g. between devices and switches) decomposition during model checking. Note that our approach consists in model checking the system model *offline*, that is, exhaustively checking all possible behaviors of the model before deployment. Third, we show that it is possible to verify complex scenarios such as to discover a network slice isolation violation. Finally, we propose a method and related architecture to avoid the issue previously discovered and verify that such a method is valid. After going through state-of-the-art in Section II, we describe the SDN platform of [1] taken as a study object in Section III. In Section IV we develop a complete formal model of the studied platform in the form of automata. An implementation of such automata is also proposed in Section V using a tool called Spin [2]. In the same section, we argue that compositional reasoning is well appropriate for SDN and propose the application of such reasoning to our previous formal model. Most of experimental works are carried out in Section VI where we discuss on performance of different verification methods according to model sizes while assuming devices are not moving (i.e. no mobility condition). The model with device mobility in Section VII allows us to pinpoint a network slice isolation issue. We finally propose a solution to correct the isolation violation and formally verify that the solution is valid. Due to page limitation, we provide with full model and methodology descriptions at a public-access repository [3] and a long version of the paper at [4].

II. RELATED WORK

The closest to our work is that of [5] which presents a tool called Kuai. The latter applies optimizations based on partial-order reduction to simplify models of SDN protocols, and presents several model checking benchmarks to assess the performance of its optimizations. The tool translates formal models given in the Murphi input format and uses PReach [6] for distributed model checking. One of its optimizations is based on *barriers* which are used to ensure the order in which switches implement SDN rules. We do not consider barriers in our platform since our SDN rules are applied asynchronously. Also, Kuai uses $(0, \infty)$ -abstractions on packet queue contents which consist in storing whether a given packet is absent (0) or present an unknown number of times (∞). Since we present

an abstract model by focusing on particular packets, we do not use this abstraction in our work. Our entire SDN platform is an asynchronous distributed system where various components are with finite number of FIFO queues. There is no priority on queues while reading incoming messages. Instead, each component reads messages by performing queue polling in a round robin fashion. We can model the system mathematically as a synchronous one by encoding state space along with current queue message contents. With regards to experimental tools, we have selected a tool called as SPIN (Simple Promela INterpreter) which has the advantages of implementing in-build partial order techniques. It spares modelers from having to write specific codes to implement partial order techniques while specifying the behaviors of the SDN platform. As per our knowledges, the use of compositional reasoning for SDN protocols is new - e.g. compared to [5]. Moreover, we consider the SDN architecture with a management plane which renders the system to be analyzed more comprehensive and more complex.

Several other works consider the formal analysis of SDN protocols. [7] provides a static analysis of header spaces which detect inconsistencies in routing tables, but does not consider more dynamic behaviors and complex issues due to interleavings as it is done in [5] and in our work. A similar approach was adopted in [8]. A modeling language called FlowLog tailored for SDN protocols was given in [9] in which the authors present model checking experiments with the SPIN model checker [2] but no particular optimizations are presented. In [10], model checking and symbolic execution are combined in order to check OpenFlow applications for a given number of packets. [11] presents data and network abstractions applied on SDN models combined with a manual refinement process based on *non-interference lemmas*.

In [12], the authors develop a framework where the system is modeled using first-order logic, and user-provided inductive invariants are checked to prove correctness. This allows one to check the system for all network topologies, and for an unbounded number exchanged packets.

The approach developed in [13] consists in checking the effect of rule updates in real-time, without affecting the system performance. This is complementary to *offline* verification approaches which analyze the system globally before execution.

In [14], reactive synthesis is used to automatically synthesize network update rules in response to network requests. Other works consider the synthesis of update rules that makes sure that the intermediate configurations during the update are all admissible [15], [16], or compute correct network-wide configurations [17].

Abstractions and other transformations that preserve the properties of networks for rendering the verification more efficient are presented in [18]. In this work, we are interested to provide the methodology to use compositional reasoning to verify SDN platform correctness. Properties selected for verification are estimated by the authors as critical for a good behavior of an SDN platform. Naturally, compositional reasoning is a well-adapted verification analysis for SDN,

since SDN platforms consist in a clear separation between control, management and data planes. In order to ensure that the SDN platform satisfies global properties, it is sufficient to verify that each plane satisfies specific local properties, which makes the model checking procedure scalable with respect to time and space complexity. This methodology also allows for analysing and designing successively each individual component of whole complex system.

III. THE SDN PLATFORM DESCRIPTION

In this document, we build a formal and abstract model of the SDN platform [1] which is taken here as a study object. We think that this SDN platform has a representative architecture to make our study as generic as possible.

A. Architecture Building Blocks

As most of SDN platforms, SDN platform [1] consists of three layers, namely a management plane, a control plane and a data plane. The management plane are formed of entities called 'managers'. Each manager has the role of interfacing with the user and of converting the user's intents into high level network policies. The control plane consists of entities called 'controllers' which have the role of converting managers' high level network policies into fine grain network rules and of enforcing those rules onto the data plane. The data plane consists of network elements embodied in the selected platform as Open vSwitches (OVS) [19]. OVSeS are open-source OpenFlow switches [20] controlled by SDN controllers.

The platform is made of several administrative domains with each domain built up with one manager, one controller and several switches (i.e. OVSeS). A given switch can only be part of one unique domain so that switches of all domains form a partition of the overall data plane. In order to simplify the controller algorithm in the forwarding of multicast/broadcast messages, the data plane topology of each domain is loop-free (the LLDP and the spanning-tree mechanisms should be added to the controller logics if non loop-free topologies are considered. This is part of our next steps). In this document, such a topology is a tree topology departing from a root switch. The latter is the domain border switch. Root switches of all domains are interconnected together in a full meshed topology so that there is only one network hop between any pairs of root switches. Figure 1 provides us with a schematic representation of the SDN platform.

The data plane is also completed with devices. Each of them can be connected to a switch port called as an *Access-port* via a network link called as an *Access-link* (thin green links). Similarly, switches within a domain are connected together via network links called as *Intra-links* (medium-size green links) thanks to their *Intra-ports*. Root switches from different domains are interconnected together via *Inter-links* (thick green links) thanks to their *Inter-ports*.

B. The User's Intent

The user's intent consists in a set of predicates on device characteristics or device capabilities [21]. This set of predicates or intent allows the user to select devices and gather

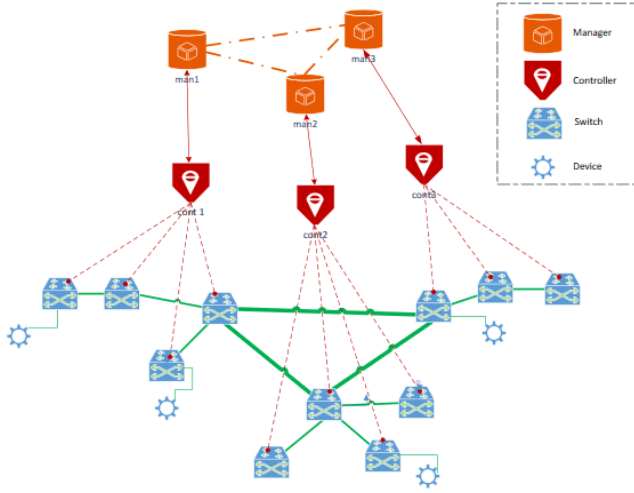


Fig. 1. SDN Platform Description

them into a private group called in the following sections as *Virtual Space* or *VS*. Devices of the same *VS* are to be connected together by controllers on the different domains where these devices are detected (i.e. via the MAC learning process). A *VS* is enforced by the control plane as a network slice in the data plane. While enforcing OpenFlow rules, controllers should make sure that a pair of devices which do not belong to any common *VS* are not connected together. This is defined as the network slice isolation property or isolation property for short (i.e. data traffic isolation between network slices) in this document. For the sake of simplicity, the user's intent will be presented in the rest of the document as a nominative list of devices - e.g. $\{d1, d2, d3\}$. This is done without losing the genericity of our discussions since the present verification work is not focusing on the consistency of the overall users' intents but rather on the safety of the underlying SDN mechanisms. Moreover, we do not present details on how the controller which identifies a given device by its MAC address, and the associated manager which identifies the same device by its name, agree with each other on the device identity, as per the limited size of the document. Please refer to [1] for further information.

C. Device Discovery via MAC Learning

The controller implements the well-known MAC learning mechanism as per [22]. Thanks to this mechanism, the controller can discover newly connected device (e.g. new MAC address) and inform the associated manager. In return, it receives from the manager high level network policies (derived from users' intents) regarding the new device.

Once the new device has been discovered by the domain controller, the latter 'leaks' multicast/broadcast packets from the former to all neighbor domains where devices of the same *VS* are located (and only to those domains). Such packets are treated as MAC learning packets in such neighboring domains, meaning that they are flooded within the aforementioned domains except to their *Access-ports*. The leaking process is

triggered thanks to knowledges coming from managers which synchronize between them, *VS* predicate and device location. In order to avoid forwarding loops during the leaking process, controllers apply a simple algorithm known as the *split horizon* algorithm. Within such an algorithm, a root switch never forwards to other root switches a packet it has received from another root switch. The *split horizon* algorithm does properly ensure reachability between domain given that we have a full mesh topology between root switches.

The work in this document is modeling the above device discovery procedure through MAC learning at all planes.

D. Packet Forwarding

A unicast packet with source MAC address 'S' and destination MAC address 'D' is forwarded if MAC addresses 'S' and 'D' are considered as part of the same *VS* (assuming the right mapping between MAC addresses and device names as discussed previously). Otherwise, the packet is dropped.

Concerning multicast (including broadcast) packets, the controller derives a multicast tree for each source MAC address 'S' [23] based on the different *VSe*s this MAC address 'S' belongs to. In order to avoid network loops when forwarding multicast packets over *Inter-links*, each domain controller implements the well-known *split horizon* algorithm taking into account the full mesh nature of the *Inter-links* topology. The *split horizon* algorithm simply consists in not forwarding to other *Inter-links* a packet coming from an *Inter-link*.

Apart from multicast packet forwarding, all the data plane including unicast forwarding and flooding mechanism which is part of the MAC learning process is modelled during this work. As it concerns the control plane and the management plane, their essential behaviors are completely captured by the models described in this document.

IV. AUTOMATA-BASED FORMAL MODELS

We formalize our model using finite automata. We start by introducing the formalism, and then describe the formal model of the SDN platform.

A. Preliminaries

Definition IV.1. Automaton: an automaton is a tuple of the form $(Q, q_{init}, \Sigma \cup \{\tau\}, \Delta, L)$, where Q is the finite set of states, $q_{init} \in Q$ is the initial state, $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ is the alphabet and τ a distinguished symbol for internal transition, $\Delta : Q \times \Sigma \cup \{\tau\} \rightarrow Q$ is the transition relation, and $L : Q \rightarrow 2^{AP}$ is a labeling function with AP a given set of atomic predicates.

A transition $\delta = (q, \sigma, q') \in \Delta$ will also be written as $q \xrightarrow{\sigma} q'$ if δ is clear from the context. Moreover, for better readability, we will use $\sigma!$ if $\sigma \in \Sigma_{out}$, and $\sigma?$ if $\sigma \in \Sigma_{in}$. This is only to help the reader; formally the label does not contain the symbols $?$ and $!$.

Definition IV.2. Synchronized product of automata: given automata $\mathcal{A}_i = (Q^i, q_{init}^i, \Sigma^i \cup \{\tau\}, \Delta^i, L^i)$ for $i \in \{1, \dots, n\}$, the synchronized product of \mathcal{A}_i is defined as $\mathcal{A} = \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n = (\Pi_{i=1}^n Q^i, (q_{init}^1, \dots, q_{init}^n), \Sigma \cup \{\tau\}, \Delta, L)$ where $\Sigma =$

$\cup_{i=1}^n \Sigma_i$, $L : \prod_{i=1}^n Q^i \rightarrow (2^{AP_1}, \dots, 2^{AP_n})$ s.t $L(q_1, q_2, \dots, q_n) = (L^1(q_1), \dots, L^n(q_n))$, and Δ is defined as follows. For all states $(q_1, \dots, q_n) \in \prod_{i=1}^n Q^i$,

- for all $1 \leq i \leq n$, and $q'_i \in Q^i$ such that $\Delta^i(q_i, \tau, q'_i), ((q_1, \dots, q_i, \dots, q_n), \tau, (q_1, \dots, q'_i, \dots, q_n)) \in \Delta$,
- for all $\sigma \in \Sigma$, writing $I_\sigma = \{1 \leq i \leq n \mid \exists q'_i \in Q^i, \Delta^i(q_i, \sigma, q'_i)\}$, $(q_1, \dots, q_n) \xrightarrow{\sigma} (q'_1, \dots, q'_n)$ if, and only if for all $i \in I_\sigma$, $\Delta^i(q_i, \sigma, q'_i)$, and for all $i \notin I_\sigma$, there exists no q'_i such that $\Delta^i(q_i, \sigma, q'_i)$.

B. Important Concepts

An instance of the platform is defined by providing the topology for different planes. Recall that there is exactly one manager associated to each controller. Each controller only communicates with its own manager and the switches in its domain. Managers are interconnected together via a complete graph of communication.

Formally, consider a set of controllers $\text{Cont} = \{c_1, \dots, c_k\}$, a set of managers $\text{Man} = \{\text{man}_1, \dots, \text{man}_k\}$ with the same cardinal, and a set of switches $\text{Sw} = \text{Sw}_1 \cup \dots \cup \text{Sw}_k$ given as a partition. Within the selected platform, the switches of Sw_i belong to the *domain* i which is controlled by the controller c_i . The latter is itself managed by man_i . We consider a set Dev of devices which can connect to switches of different domains.

In each domain i , there is one designated switch called the *root switch*, denoted by $\text{root}_i \in \text{Sw}_i$. The data plane topology is a graph $G = (\text{Sw}, E)$ such that the subgraph restricted to each Sw_i is an undirected tree rooted at root_i , and the subgraph induced by the set of roots $\{\text{root}_1, \dots, \text{root}_k\}$ is a complete graph.

In our studied platform, we distinguish so-called *Access-ports* through which devices connect to switches. We also have *Intra-ports* which are used to interconnect switches that belong to the same domain, and *Inter-ports* which are used to interconnect *root switches* of different domains. We also define a finite set Pts which refer to the set of *Access-ports* available at all switches.

Platform instance: an instance of the studied platform is defined as a tuple $(\text{Man}, \text{Cont}, \text{Sw}, G)$ where:

- $\text{Man} = \{\text{man}_1, \dots, \text{man}_k\}$,
- $\text{Cont} = \{c_1, \dots, c_k\}$,
- $\text{Sw} = \bigcup_{i=1}^k \text{Sw}_i$,
- $G = (\text{Sw}, E)$ is the topology graph as explained above.

We also use the function $\text{cont} : \text{Sw} \rightarrow \text{Cont}$ which identifies the controller associated to each switch, thus determining its domain: for all $i \in \{1, \dots, k\}$ and $v \in \text{Sw}_i$, we have $\text{cont}(v) = c_i$.

Virtual Spaces (VSes): we consider the covering of Dev by the set of *Virtual Spaces* $\text{VS} = \{V_1, V_2, \dots, V_f\}$, with $V_i \subseteq \text{Dev}$ for all $1 \leq i \leq f$, and $\bigcup_{i=1}^f V_i = \text{Dev}$. Let us define the function $\text{VS}(x) = \{V \in \text{VS} \mid x \in V\}$ which assigns to each device x the set of VSes to which x belongs.

Exchanged Data Packets: two types of packets are exchanged between switches and devices. *MAC learning packets* are sent by devices and forwarded by switches, and they

contain the device identifier; while *ping packets* contain the source and target device identifiers.

These are defined, respectively, as $\text{MacPkts} = \{\text{mac}\} \times \text{Dev}$ and $\text{PingPkts} = \{\text{ping}\} \times \text{Dev} \times \text{Dev}$. These packets will be written as $\text{mac}(x)$ and $\text{ping}(x, y)$ respectively, for $x, y \in \text{Dev}$.

Exchanged Control Packets: let us define $\text{ManPkts} = \text{Dev} \times \{1, \dots, k\} \times 2^{\text{VS}}$ a set of packets from the management plane to the controller plane and $\text{ContPkts} = \{(\text{man}_i, c_i) \mid i \in \{1, \dots, k\}\} \times \text{Dev}$ is the set of packets from the controller plane to the management plane.

It is noted that MacPkts forwarded by switches to the associated controllers are also part of exchanged control packets.

OpenFlow Rules: controller sends two types of rule updates to switches.

Rules impacting the forwarding of ping packets are defined as $\text{PingRules} = \text{Dev} \times \text{Dev} \times (\text{Pts} \cup \text{Sw})$ where the triple contains the source and target devices identifiers and the port or the switch to forward to.

Rules impacting the forwarding of MAC learning packets are $\text{MacRules} = \text{Dev} \times \text{Sw}$ where the pair contains the identifier of the device that has generated the mac learning packet, and the switch to forward to.

We let $\text{Rules} = \text{PingRules} \cup \text{MacRules}$.

The automata for devices, switch, controller and manager are denoted as \mathcal{A}^{Dev} , \mathcal{A}^{Sw} , $\mathcal{A}^{\text{Cont}}$ and \mathcal{A}^{man} . Detailed formal definitions of the automata together with different SPIN implementation source codes are provided in a public-access repository [3].

V. FORMAL VERIFICATION

During the following work, we use linear temporal logic (LTL) [24] to specify different system properties including the ones that we want to verify. LTL is build over propositional logic (based on Boolean operators \neg, \vee, \wedge) by adding the notion of discrete (*i.e* natural number) timeline and temporal operators such as the following $\{\Box, \Diamond\}$ just to name the ones we use here: $\Box\psi$ means ψ is **True** in **all** future moments, $\Diamond\psi$ means ψ is **True** in **some** future moments.

LTL formulas are built over a finite set of propositional variables referred to as *Atomic Propositions* (AP) combined with aforementioned Boolean and temporal operators. Temporal operators can be nested. For instance $\Box\Diamond\phi$ refers to executions in which at all moments, there is a moment in future where ϕ holds; thus, ϕ holds at infinitely many points in time. Formula $\Diamond\Box\phi$ refers to executions in which there is a moment in time after which ϕ holds at all positions; thus ϕ becomes an invariant after a while. We use the *finite trace* semantics of LTL [25] without next operators denoted by $\text{LTL}_f \setminus X$.

A. Property Specifications

We verify two important properties of our platform. The first is a *safety* property, called as the *isolation* property. It states that only devices that belong to a common VS can exchange ping packets and a device not belonging to a particular VS

cannot eavesdrop ping packets exchanged among that VS group. This can be expressed in LTL as follows.

$$\text{Isolation} = \Box \left(\bigwedge_{\substack{x,y \in \text{Dev}, \\ \text{VS}(x) \cap \text{VS}(y) = \emptyset}} \text{ping}(x,y) \notin \text{rpings}_y \right).$$

This LTL formula thus describes the set of executions which never enter a state where a packet $\text{ping}(x,y)$ with $\text{VS}(x) \cap \text{VS}(y) = \emptyset$ is received, and no device can eavesdrop the packet which as to be received by different devices.

The second property is called as the *connectivity* property. It states that whenever a ping packet is sent to a device that belongs to a same VS group, the packet is eventually received; however this property only holds after some point in time, that is, when the MAC learning algorithm has finished. So the SDN platform eventually allows a common VS group to exchange the data packets among themselves. This can be expressed in LTL as follows:

$$\text{Connect} = \bigwedge_{\substack{x,y \in \text{Dev} \\ \text{VS}(x) \cap \text{VS}(y) \neq \emptyset}} \Diamond (\text{ping}(x,y) \in \text{spings} \wedge \Diamond \text{ping}(x,y) \in \text{rpings}_y).$$

In our work we consider two scenarios. In the *no-mobility* scenario, devices do not change their respective position once they are connected to an *Access-port*. In the *mobility* scenario, devices can change their position at any time.

The first scenario called as Nomobility can be expressed in LTL as follows:

$$\text{Nomobility} = \bigwedge_{\substack{x \in \text{Dev}, v \in \text{Sw}, \\ p \in \text{Sw} \cup \text{Pts}}} \Box ((v,p) = \text{pos}(x) \rightarrow \Box (v,p) = \text{pos}(x))$$

B. Compositional Reasoning

We present here a compositional reasoning methodology which allows one to model check a system by decomposing the problem into simpler verification tasks on each of its components. Such compositional techniques appeared early in the model checking community [26]–[28]. We also refer the reader to [29] where existing rules and algorithms are summarized.

Given an automaton \mathcal{A} and LTL formulas ϕ, ψ , let the triple $\langle \phi \rangle \mathcal{A} \langle \psi \rangle$ denote $\mathcal{A} \models \phi \rightarrow \psi$. Intuitively, ϕ represents the *assumption* we make on the environment of \mathcal{A} , while ψ is the *guarantee* that \mathcal{A} gives provided that the environment satisfies ϕ . We state the main theorem we use as follows.

Theorem V.1. Consider automata $\mathcal{A}_1, \mathcal{A}_2$ with atomic propositions AP_1, AP_2 and alphabets Σ_1, Σ_2 , and $\text{LTL}_f \setminus X$ formulas ϕ_1, ϕ_2, ϕ_3 with $\Sigma_{\phi_1} \subseteq \Sigma_1$, $\Sigma_{\phi_2} = \Sigma_1 \cap \Sigma_2$, and $\Sigma_{\phi_3} \subseteq \Sigma_2$. We have the following rule.

$$\frac{\langle \phi_1 \rangle \mathcal{A}_1 \langle \phi_2 \rangle \quad \langle \phi_2 \rangle \mathcal{A}_2 \langle \phi_3 \rangle}{\langle \phi_1 \rangle \mathcal{A}_1 \parallel \mathcal{A}_2 \langle \phi_3 \rangle}.$$

Thus, in order to prove $\langle \phi_1 \rangle \mathcal{A}_1 \parallel \mathcal{A}_2 \langle \phi_3 \rangle$ it suffices to find a formula ϕ_2 and prove the two assertions $\langle \phi_1 \rangle \mathcal{A}_1 \langle \phi_2 \rangle$ and $\langle \phi_2 \rangle \mathcal{A}_2 \langle \phi_3 \rangle$ as per the previous theorem.

A proof of this theorem for *safety* properties can be found

in [29]. Our formulas are general $\text{LTL}_f \setminus X$ formulas in the finite trace semantics. However, in our model of the SDN protocol, all executions eventually stop so that we can encode all our formulas as safety conditions using the Promela keyword `timeout` (which detects the end of an execution). For completeness, we do provide a proof of this theorem for general $\text{LTL}_f \setminus X$ formulas which can be applied to general models. This proof can be found together with the detailed formal automata definitions at [3].

We argue that compositional reasoning is particularly well adapted for verifying SDN protocols. In fact, SDN platforms are distributed systems made of well-separated components, which are the management plane, the controller plane and the data plane. In order to meet our global specifications of isolation and connectivity, each plane has to satisfy a certain property. One of the sources of complexity in verification comes from the fact that several planes are considered in the model. Compositional reasoning allows us to verify properties of each plane separately, and combine these rules to infer the specification for the global system. We explain the application of Theorem V.1 in the next section.

To use Theorem V.1 we need to formally state the specifications for each components in LTL. Finding these properties requires domain knowledge. In fact, one needs to understand with which intentions the system was designed in order to write these formulas.

The management plane and control plane should work together to satisfy the following requirement:

$$l_1 = \bigwedge_{\substack{1 \leq i \neq j \leq k, x, x' \in \text{Dev} \\ \text{VS}(x) \cap \text{VS}(x') \neq \emptyset \\ v \in \text{Sw}_i, v' \in \text{Sw}_j \\ p, p' \in \text{Pts}}} \Box (\text{pos}(x) = (v,p) \wedge \text{pos}(x') = (v',p') \wedge (\text{rpos}_i(x, \text{root}_i) \neq \perp) \rightarrow \Diamond \text{Sync}(c_i, \text{root}_i, (x, \text{root}_j))).$$

The above formula says that whenever devices x and x' are respectively connected at switch-port pairs (v,p) and (v',p') in respectively domains i and j , if the root switch of domain i has received a MAC learning packet originated from device x , then eventually the root switch will receive a MAC update rule from the controller to forward MAC packets of device x towards root_j . This l_1 captures the fact that the control plane sends appropriate MAC rule updates to the root switch of its domain, thus to the data plane.

$$l_2 = \bigwedge_{\substack{1 \leq i \leq k, x, x' \in \text{Dev} \\ \text{VS}(x) \cap \text{VS}(x') \neq \emptyset \\ v, v' \in \text{Sw}, p, p' \in \text{Pts} \\ u \in \text{Sw}_i, q, q' \in \text{Pts} \cup \text{Sw}}} \Box (\text{pos}(x) = (v,p) \wedge \text{pos}(x') = (v',p') \wedge (\text{rpos}_i(u, x) = q) \wedge (\text{rpos}_i(u, x') = q') \rightarrow \Diamond \text{Sync}(c_i, u, (x, x', q'))).$$

The above formula states that, given devices x, x' connected at (v,p) and (v',p') respectively, if another switch u has received and forwarded MAC learning packets to the controller witnessed by $(\text{rpos}_i(u, x) = q) \wedge (\text{rpos}_i(u, x') = q')$, then a ping update rule will eventually be sent to switch u to forward packets $\text{ping}(x, x')$ to q' . Thus, l_2 expresses that the control

plane sends appropriate ping rule updates to all the switches and thus to the data plane of its domain.

The above requirements should be satisfied jointly by the management and control planes, that is, we would like to establish that $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$ should be true. However, this check can still be costly for large topologies. Therefore, we apply again Theorem V.1 to check this compositionally.

We need to introduce another intermediate formula I_3 between the management plane and controller plane. This captures that whenever a management node man_i receives information about a device's position from its controller c_i , eventually, all management nodes man_j forward this information to their respective controller c_j . Thus, the information about a device of domain i is eventually shared with the controller of other domains j .

$$I_3 = \bigwedge_{\substack{1 \leq i \leq k, x \in \text{Dev} \\ j \in [1, \dots, k]}} \square((\text{dinfo}_i(x) \neq \perp) \rightarrow \Diamond \text{Sync}(\text{man}_j, c_j, (x, \text{dinfo}_i(x), \text{VS}(x))))$$

So we can check the satisfaction of $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$ by verifying $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \langle I_3 \rangle$, and $\langle I_3 \rangle \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$. Complete description of experimental details are provided in [3].

VI. NO-MOBILITY SCENARIO VALIDATION

Let us call the Isolation, Connect, and Nomobility mentioned in section 4.4 as ϕ_I , ϕ_C and ψ respectively. Recall the intermediate formulas l_1 , l_2 , and l_3 . We are going to verify that the specified SDN platform described in section 3 and modeled in section 4 satisfies the Isolation and Connect properties under the assumption Nomobility.

We are going to compare the usage of Compositional Reasoning (CR) in verifying the SDN platform over monolithic approach. Monolithic approach consists in verifying the entire system i.e checking whether $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$ is true or false.

Compositional rule method 1 (CR method 1) consists in splitting the control plane from the data plane and thus in verifying separately the two automata: $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$, and $\langle l_1 \wedge l_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$. Compositional rule method 2 (CR method 2) consists in further splitting the management plane from the control plane and checking three triples: $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \langle l_3 \rangle$, $\langle l_3 \rangle \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$, and $\langle l_1 \wedge l_2 \rangle P_D \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$.

The results for the three methods applied to various SDN platform sizes are shown in Table I. For a given size of the SDN platform, we checked all possible SDN constrained topologies as mentioned in Section 3 and 4 and for all possible random choices carried by the devices to connect to their *Access-port* positions. This is the main reason for large memory usage and CPU time spent as indicated in Table I. Within the Nomobility condition, once the device has selected its position and has sent its first MAC learning packet, it waits for a timeout. After the first timeout, it sends a second MAC learning packet and waits for the second timeout. After the second timeout, it sends ping packets to all other devices. In this scenario we check that devices which are sharing a

common VS can receive ping packets from each other (i.e. Connect property), and that devices which are not belonging to this particular VS cannot receive ping packets from the group and vice versa, and that a device cannot eavesdrop ping packets intended for other devices (i.e. Isolation property).

n domains, (X,Y,Z) OVSes	Monolithic Method		CR Method1		CR Method2	
	CPU time	RAM used	CPU time	RAM used	CPU time	RAM used
1, (1, -, -)	0.38s	133MB	0.85s	134MB	1.11s	135MB
1, (2, -, -)	166s	723MB	1.05s	135MB	1.32s	135MB
1, (3, -, -)	-	-	6.8s	168MB	7s	168MB
1, (4, -, -)	-	-	2m 34s	532MB	2m 32.4s	531MB
1, (5, -, -)	-	-	39m 33s	5002MB	39m 40s	5002MB
2, (1, 1, -)	-	-	4.07s	145MB	1.66s	135MB
2, (2, 1, -)	-	-	9.5s	164MB	4.05s	164MB
2, (3, 1, -)	-	-	1m 29s	505MB	1m 10.69s	505MB
2, (2, 2, -)	-	-	47.73s	502MB	35.77s	502MB
3, (1, 1, 1)	-	-	18m 25s	4292MB	4.22s	162MB
3, (1, 1, 2)	-	-	36m 11s	6982MB	34.62s	472MB

TABLE I
MONOLITHIC VS COMPOSITIONAL APPROACHES (WITH NO-MOBILITY)

In order to test these Isolation and Connect properties we create the following set of VSes, $\text{VS} = \{ \{d_1, d_2\}, \{d_3\} \}$, where $\{d_1, d_2\}$ is one VS containing devices d_1 and d_2 and $\{d_3\}$ the other VS containing a single device d_3 . We verify that d_3 cannot receive ping packets from devices in $\{d_1, d_2\}$ and vice-versa in order to assert that the SDN platform satisfies the Isolation property. We also verify that eventually d_1 and d_2 can receive ping packets of one another to assert that the SDN platform satisfies the Connect property. For the particular case of no-mobility, the SDN platform satisfies both Isolation and Connect properties. To shortly capture the SDN platform architecture we adopt for the rest of the document the following notations. We denote by $n, (X, Y, Z)$ the set of topologies with n domains ($1 \leq n \leq 3$ in our case), where (X, Y, Z) refers to the number of OVSes in each domain. For the single domain with 3, 4, and 5 switches there are respectively two, four and five possible data plane topologies since we restrict to trees. For the case of two domains ($n = 2$), the possibilities are $(2, 1, -)$, $(3, 1, -)$, and $(2, 2, -)$ in this case, there are, respectively, one, two, and one possible data plane topologies. With three domains ($n = 3$), the possibility is $(1, 1, 2)$ in which case there is one possible data plane topology. In our experiments, we enumerate all topologies that match a given template (X, Y, Z) to check exhaustively all possible scenarios. Therefore, the cases with larger numbers of topologies take more time to check.

We ran our experiments on an Intel i7 processor with 13GB of available RAM. Table I compares the monolithic and two compositional approaches, called Method 1 and Method 2, with respect to CPU time and memory usage on SDN topologies of varying sizes.

While the monolithic approach fails to scale above trivial topologies, compositional methods helped us to complete the verification process up to 3 domains in our experiments. Interestingly, the minimum number of domains which allows for a good verification of loop-free split horizon algorithm (Cf. section 3) is three. CR method 1 and method 2 provide the same performance up to 2 domains. When the number of domain is 3, CR method 2 gives advantage over the RAM usage and CPU time. Within CR method 1, for 3 domains, much of its CPU and RAM resources are consumed in

checking $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$. Here the majority of CPU and RAM resources are consumed in checking $\langle l_1 \wedge l_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$. The sizes of state space produced in the monolithic method reaches the order of 10^6 for two switches in a single domain, while the compositional approaches produce around 10^4 states for the same architecture.

VII. ISSUE WITH DEVICE MOBILITY

In the previous section, we verify that the studied SDN platform does satisfy the required specifications, namely the *isolation* property and the *connectivity* property. What if we allow devices to move freely but still use the data plane service to exchange packets? Given this freedom, devices can select their new positions wherever and whenever they want.

This freedom makes the SDN system vulnerable w.r.t. data isolation property. We identify an error which consists in an execution violating the isolation property. We can produce the error simply with our monolithic model checking for one domain and two switches, that is, $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ is violated, where ϕ_I the Isolation property.

We find this error by considering $\text{VS} = \{\{d_1, d_2\}, \{d_3\}\}$ and the following mobility scenario: once first MAC learning packets are exchanged and ping rules are updated, devices d_2 and d_3 swap their respective *Access-port*. After the mobility events, Spin raises an Isolation violation reporting that device d_3 receives a ping packet send by d_1 to d_2 .

A counter-example trace from Spin reveals that this is a real-time concurrence between data packets and the related SDN rule enforcement (i.e. delay for applying SDN rules for these data packets). Thus, the issue is essentially due to the transmission delay experienced at the control plane (i.e. control order transmission latency between the SDN controller and the OVS).

A. Local Controller Solution

To solve *isolation* violation, one possible solution consists in implementing a hierarchical controller architecture with a new controller embedded together with each switch that we call as *local controller* in addition to existing centralized controller that we call as the *central controller*. Figure 2 illustrates such a controller hierarchical architecture. The *local controller* role simply consists in blocking any outgoing traffic towards a port (e.g. port 2) when the existing device (e.g. d_2) is disconnecting from this port (e.g. a port down status message sent to controllers). It maintains such a blocking rule until a new device (e.g. d_3) is connected to the port and MAC learning messages from this device are processed by the *central controller* and new forwarding rules related to the new device are received by the switch. In order to ensure new rules are actually received by the switch, the *local controller* is implemented as an OpenFlow proxy between the switch and the *central controller*. Moreover, the number of devices the *local controller* should deal with is smaller than the one the *central controller* is dealing with by the order of magnitude the number of switches. The variability of control message rate is to be smaller from the *local controller* perspective,

which leads to an easier engineering in terms of processing resources (i.e. for low-cost switches). Such a local controller can be modeled by an automata as $\mathcal{A}^{\text{loc.Cont}}$.

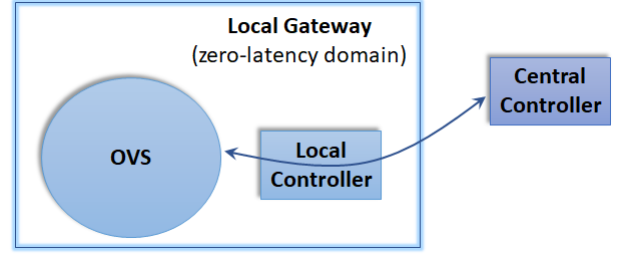


Fig. 2. Local Controller Architecture

B. Local Controller Solution Validation

In this section, we verify that the proposed local controller solution satisfies the Isolation property, i.e. $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$, where ϕ_I is the Isolation property defined in Section V-A and $\mathcal{A}^{\text{loc.Cont}}$ proposed local controller method specified previously. By using the Theorem V.1, we check $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \wedge l_2 \rangle$ and $\langle l_1 \wedge l_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ to prove $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$, where l_1 and l_2 are the intermediate formula defined in Section V-A. The proposed local controller does satisfy the required isolation property. We carry out the experiment for different topologies of the SDN platform. For each topology, we monitor the computational time and the memory usage. Table II provides a synthetic view of these experimental data.

n domains, (X, Y, Z) Switches	Compositional reasoning rule $\langle l_1 \wedge l_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_1 \rangle$ and $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle l_2 \rangle$	
	Total Computation time	Maximum RAM memory used
1, (1, -, -)	4.8sec	188 MB
1, (2, -, -)	8.4sec	224 MB
1, (3, -, -)	1min 2sec	448 MB
1, (4, -, -)	6min 22sec	982 MB
1, (5, -, -)	1hr 5min 8sec	7245 MB
2, (1, 1, -)	11sec	214 MB
2, (2, 1, -)	33sec	395 MB
2, (3, 1, -)	3min 10.3sec	865 MB
2, (2, 2, -)	96.5sec	848 MB
3, (1, 1, 1)	19min 9.8sec	4291 MB
3, (1, 1, 2)	36min 44.39sec	6981 MB

TABLE II
LOCAL CONTROLLER PROPOSAL VALIDATION USING CR METHOD
(MOBILITY SCENARIO)

VIII. CONCLUSION

In this document we have demonstrated that it is possible to fully model an SDN architecture including the entire 3 planes, namely the data plane, the control plane and the management plane. It is also possible to include device connectivity and mobility. Although state space can rapidly explode, it is still possible to validate a complex SDN architecture based on a minimum representative model of it.

Within this framework, we have shown that compositional model checking is well-adapted for verifying SDN architectures. It allows one to verify the whole system while

verifying each individual component of it. The decomposition of an SDN architecture can be performed, as an illustrative experiment, vertically between planes. This provides more scalability especially when complexity becomes high.

We have demonstrated that it is possible to verify that users' intents are correctly enforced on the data plane in the form of network slices using model checking. There is a need however to develop an automatic framework to translate high-level intents to concrete network policies in a generic way.

Finally, we have discussed on the trade off between a centralized architecture and a more distributed one and have set up an architecture proposition with regards to network slice isolation issue related to control message transmission latency. Indeed, most of SDN architecture advocates for a centralized controller as this optimizes the interaction with applications and simplifies the related algorithms. This may lead however to some important issues such as the network slice isolation violation revealed in this document. The probability of such an issue is likely to be exacerbated when operating at high data rate. A hierarchical architecture such as the one we propose can solve the issue.

In order to complete this work, we plan to introduce formal synthesis methods aiming at automatically synthesizing the local SDN controller algorithm with regards to some set of safety constraints as tackles in [30]. Other directions are to abstract the system so that traditional supervisory control synthesis of reactive systems can be applied [31].

REFERENCES

- [1] M. Boussard, D. T. Bui, L. Ciavaglia, R. Douville, M. L. Pallec, N. L. Sauze, L. Noirie, S. Papillon, P. Peloso, and F. Santoro, "Software-defined lans for interconnected smart environment," in *2015 27th International Teletraffic Congress*, 2015, pp. 219–227.
- [2] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.
- [3] A. Majith, O. Sankur, H. Marchand, D. T. Bui, *Open access Repository for "Compositional model checking of an SDN platform": Spin source codes*, <https://gitlab.inria.fr/anoordhe/drcn2021milan>.
- [4] A. Majith, O. Sankur, H. Marchand, and T. Dinh, "Compositional model checking of SDN platform," Feb. 2021, working paper or preprint, https://hal.inria.fr/hal-03153317/file/sdn_verif.pdf.
- [5] R. Majumdar, S. Deep Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *2014 Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2014, pp. 163–170.
- [6] B. Bingham, J. Bingham, F. M. d. Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *2010 Ninth Int. Workshop on Parallel and Distributed Methods in Verification, and Second Int. Workshop on High Performance Computational Systems Biology*, USA, 2010, p. 2836.
- [7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, 2012, pp. 113–126.
- [8] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, p. 290301.
- [9] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *PSACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013, p. 7984.
- [10] M. Canini, D. Venzano, P. Perešini, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *9th USENIX Conference on Networked Systems Design and Implementation*, 2012, p. 10.
- [11] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking sdn controllers," in *Formal Methods in Computer-Aided Design*, 2013, pp. 145–148.
- [12] T. Ball, N. Björner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: towards verifying controller programs in software-defined networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 282–293.
- [13] K. A., X. Zou, K. Zhou, M. Caesar, and P. Brighten Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013, pp. 15–27.
- [14] A. Wang, S. Moarref, B. T. Loo, U. Topcu, and A. Scedrov, "Automated synthesis of reactive controllers for software-defined networks," in *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, Oct. 7-10, 2013*, 2013, pp. 1–6.
- [15] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, 2012, pp. 323–334.
- [16] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," *SIGPLAN Not.*, vol. 50, no. 6, p. 196207, Jun. 2015.
- [17] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *Computer Aided Verification*, R. Majumdar and V. Kunčák, Eds., 2017, pp. 261–281.
- [18] G. Plotkin, N. Björner, N. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 16, 2016, p. 6983.
- [19] L. Foundation. (2016) Open vswitch. [Online]. Available: <https://www.openvswitch.org/>
- [20] O. N. Foundation. (Dec. 19, 2014) Openflow switch specification - version 1.5.0. [Online]. Available: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf>
- [21] P. Peloso, D. T. Bui, and M. Boussard, "Enforcing users' constraints in dynamic, software-defined networks of devices," in *2017 19th Asia-Pacific Network Operations and Management Symposium*, 2017, pp. 106–111.
- [22] L. Foundation. (2016) Open vswitch advanced features. [Online]. Available: <http://docs.openvswitch.org/en/latest/tutorials/ovs-advanced/>
- [23] D. T. Bui, R. Douville, and M. Boussard, "Supporting multicast and broadcast traffic for groups of connected devices," in *2016 IEEE NetSoft Conference and Workshops*, 2016, pp. 48–52.
- [24] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS 77, 1977, p. 4657.
- [25] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, p. 854860.
- [26] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP congress*, vol. 83, 1983, pp. 321–332.
- [27] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., 1985, pp. 123–144.
- [28] E. W. Stark, "A proof technique for rely/guarantee properties," in *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985*, pp. 369–391.
- [29] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.
- [30] G. Kalyon, T. Le Gall, H. Marchand, and T. Massart, "Symbolic supervisory control of distributed systems with communications," *IEEE Transaction on Automatic Control*, vol. 59, no. 2, pp. 396–408, February 2014.
- [31] N. Ayeb, E. Rutten, S. Bolle, T. Coupaye, and M. Douet, "Towards an Autonomic and Distributed Device Management for the Internet of Things," in *IEEE 4th International Workshops on Foundations and Applications of Self* Systems*, Umea, Sweden, Jun. 2019, pp. 246–248.